OOP
Reflection

Kasper Østerbye
Mette Jaquet
Carsten Schuermann
IT University Copenhagen

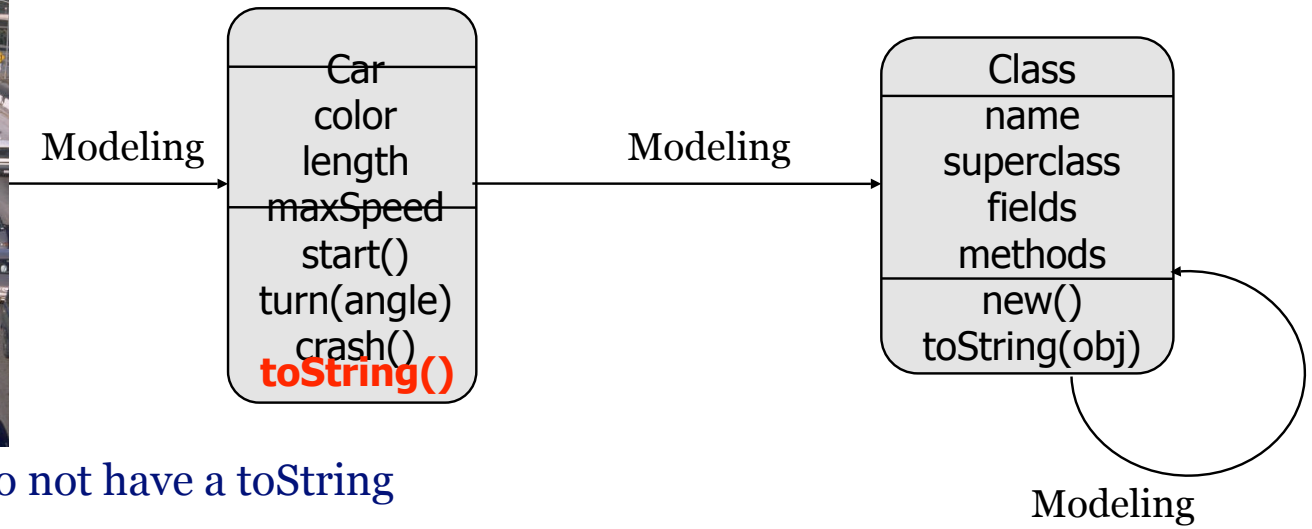
Today's schedule

- Reflection
  - modelling a domain vs. modelling objects
- Testing
  - Example: Checking that all fields are private and have a getter method
  - A unit testing framework
- Object Irrelevance
- Annotations
  - Adding metadata to programs
- Summary

Philosophically aside

The objects in the application models aspects in the application domain.

E.g.



But real cars do not have a toString method.

•The solution is to model the classes too.

•A class whose instances are classes, is called a *metaclass*.

- The program contains not only a model of the application objects, but *also* a model of itself, which is why it is called reflective

From a symmetry argument, there has to be a model of the model of the model etc. This is often 'solved' by a loop.

Concepts

A program which manipulates or examines itself is called *reflective*.

- *Introspection* means examining properties of the program
- *Intercession*/Manipulation, means changing properties of the program

- *Structural* reflection means to have access to the structure of the program, e.g. classes, methods, fields.
- *Behavioural* reflection means to have access to the run-time aspects, e.g. how objects are created, how methods are called.

- Reflection can take place at compile-time, load-time, or run-time.

- Traditionally one has in OOP only considered full reflection, that is both introspection and intercession, and both structural and behavioural, and at run-time.

- Nobody has ever succeeded in making full reflection efficient.

Usages of reflection

- To find out which methods a class has at run-time
  - Unit testing
  - Debugging
  - Plug and play systems
- Dynamic loading and maintenance of running systems
- Enterprise beans
- General storage and transmission of objects
- Redefining how method invocation is done, to achieve distribution.
- Transparent persistence.
- Programming language adaptation
- Object browsers
- Semantic checkers

In Java we can navigate the class structure using the java.lang.reflect package, and the class Class from java.lang.

Some of the methods of Class are:

getClass() returns the class for any object.

getDeclaredFields() returns an array of all fields in a class.

getDeclaredMethods() returns an array of all methods in a class.

newInstance() creates an instance of the class.

getInterfaces() returns an array of interfaces

Some of the methods of Method are:

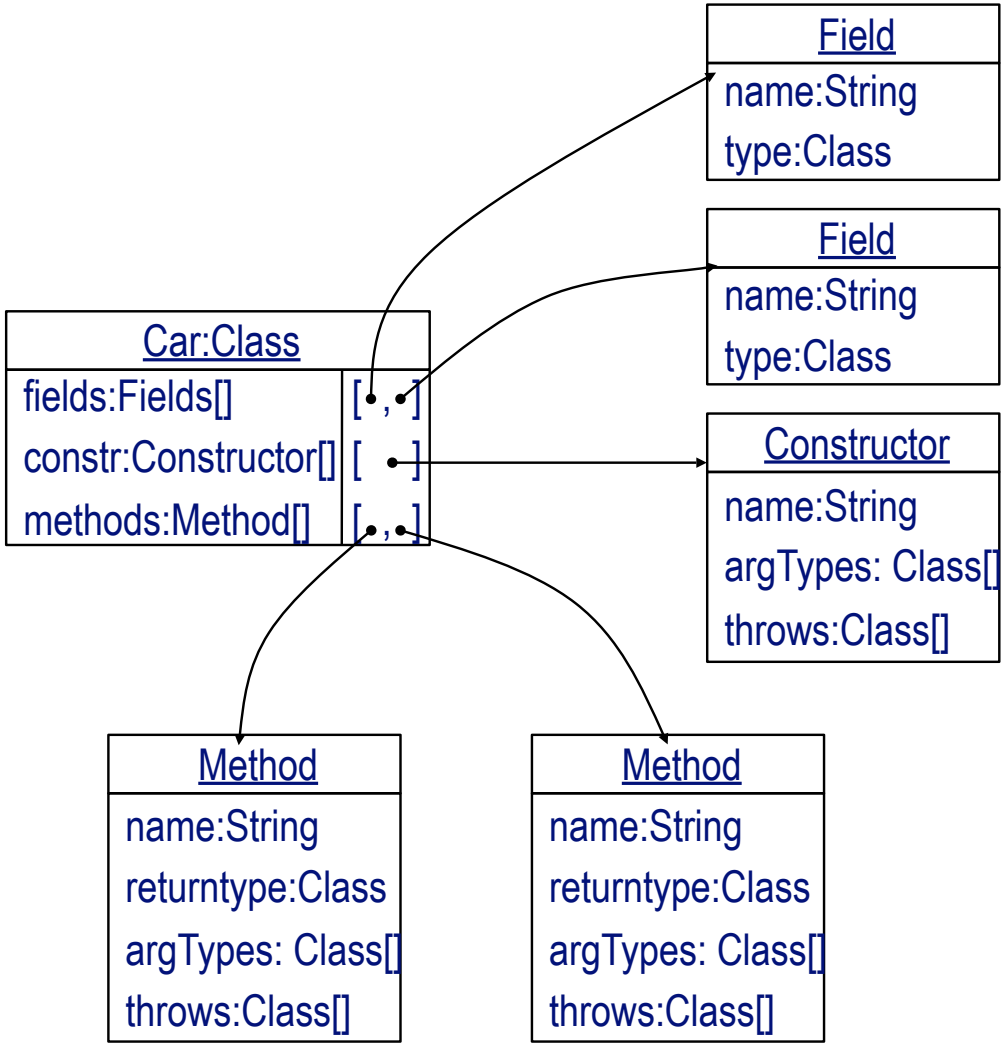
getParameterTypes() returns an array of Class

getModifiers() returns an integer representing which modifiers it has.

invoke(Object, Object[]) calls the method.

The meta model – an example

```
class Car{
    private Person owner;
    private String make;
    public Car(String make, Person owner){
        this.make = make;
        this.owner = owner;
        owner.setCar(this);
    }
    public void setOwner(Person owner){
        this.owner.setCar(null);
        this.owner = owner;
        owner.setCar(this);
    }
    public String toString(){
        return make +" owned by " + owner;
    }
}
```



Encapsulation checker

Assume we have a rule that all fields should be private, and they should have a public getter.

Write a program that gives an error for each field where this rule is broken.

The program to the right shows the outline of the program.

```
import java.lang.reflect.*;
class EncCheck {
    StringBuffer errorLog = new StringBuffer();
    Class clazz; // The class we are examining
    public void check(Class c)
        throws Exception {
        clazz = c; //… do the check and print out errors
    }
    public static void main(String[] args)
        throws Exception {
        String className = args[0];
        Class c = Class.forName(className);
        EncCheck checker = new EncCheck();
        checker.check(c);
    }
}
-----
dos:>java EncCheck dk.oop.itu.lecture6.MyClass
```

Finding all fields

The first thing we need to do is to find all fields in the class, and make sure they are private.

Does the rule apply to static fields as well? We will ignore static fields.

The underlined aspects are part of the reflection package, the rest is plain java.

```
List<Field> nonStaticFields;


public void findAllNonStaticFields(){
    Field[] fields = clazz.getDeclaredFields();
    nonStaticFields = new ArrayList<Field>();
    for (Field f : fields){
        int modifiers = f.getModifiers();
        if (! Modifier.isStatic(modifiers) )
            nonStaticFields.add(f);
    }
}
```

Checking that all non-static fields are private

This is a combination of using an iterator, and the somewhat funny way to check for the modifiers.

Alternatively, the check could be written in one line:

```
Modifier.isPrivate( f.getModifiers() )
```

Some find that more succinct

```
public void ensurePrivate() {
    for ( Field f : nonStaticFields ){
        int modifiers = f.getModifiers();
        if (! Modifier.isPrivate(modifiers) )
            errorLog.append
                (f.toString() + " is not private\n");
    }
}
```

Check that there are get accessors for each

Here we use the method getMethod on Class, which finds a method based on the signature of the method.

The signature of a method is its name, and the type of the parameters.

We follow the principle of accessors having the same name as the field, but prefixed by ”get”, and the first letter of the field name as upper case.

E.g. if the field is named foo, the getter is named getFoo.

```
private static final Class[] noParameters = new Class[0];


public void ensureGetters(){
    for ( Field f : nonStaticFields ){
        try{
            clazz.getMethod(getter(f), noParameters);
        }catch(NoSuchMethodException ex){
            errorLog.append
                (f.toString() + " has no accessor\n");
        }
    }
}
private String getter (Field f){
    String name = f.getName();
    String first = name.substring(0,0).toUpperCase();
    String rest = name.substring(1,name.length());
    return "get"+ first + rest;
}
```

Specifying signatures

The method getMethod() takes as argument the name of the method, and an array of classes, which specify what parameters the method expects.

Assume we have a method

```
foo(boolean b , Person p, int[] i)
```

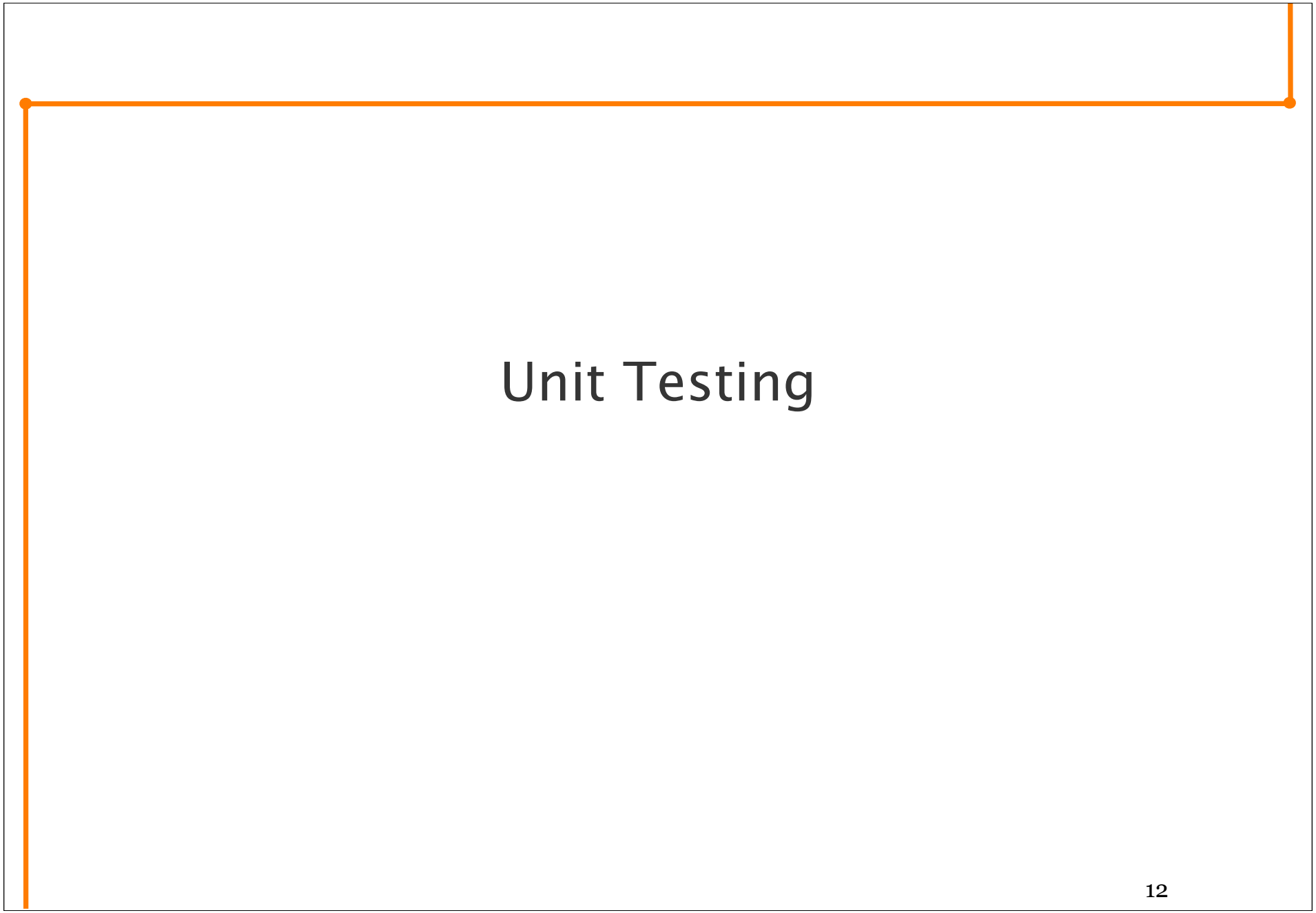
The parameters should be written up as the elements in an array

Such an array can be written as:

```
Class[ ] fooParam = new Class[ ] {boolean.class, Person.class, int[].class};
```

In general, one can obtain a class object for a given class, interface or primitive type as

```
type.class
```


Unit Testing

Test Driven Development

Unit testing is one of several core practices of the eXtreme Programming (XP) methodology.

Using a testing framework like JUnit, tests for all classes in a system can be collected in a test suite, and all tests can be run and evaluated by a push on a button.

Some benefits of unit testing:

- Quick feedback
- Automated validation of testresults
- Confidence in the code
- Collective code ownership
- Reuse of tests

Unit Testing

The principle is that together with the development of a class, one should also write a set of tests, to ensure that the class works as expected.

Unit testing step by step:

1. Write one test.
2. Compile the test. It should fail, as you haven't implemented anything yet.
3. Implement just enough to compile.
4. Run the test and see it fail.
5. Implement just enough to make the test pass.
6. Run the test and see it pass
7. Repeat from the top.

Test framework

A framework to be used for performing unit tests can be implemented using reflection.

The next slides are about the development of such a system and how to write the tests.

A typical test class might look as the class to the right.

Each method that is prefixed by ”test” should be executed. All failures are collected and printed in the main method.

```
public class MyTestClass extends TestClass
{   public void testParseInt(){
        check("'123' is 123",Integer.parseInt(”123”)==123 );
    }
    public void testParseDouble(){      //Should fail
        check("15 is 12.5",Double.parseDouble(”15”) == 12.5 );
    }
    public void testStringEquality(){
        String s = "abc";
        check("abc is aBc", s.equalsIgnoreCase("aBc"));
    }
}

// main method in some class – does not matter which
public static void main(String[] args){
    System.out.println(
    UnitTester.testTheClass("dk.itu.oop.lecture6.MyTestClass"));
}
```

The TestClass

This is possibly the simplest TestClass which can be used.

It defines two check methods, each taking a boolean condition expected to be true. One also has a String parameter describing the test.

In addition a CheckFailure exception is declared. If a check fails, this exception is thrown.

```
public class TestClass {

    public class CheckFailure extends RuntimeException {
            public String msg;
            CheckFailure(String msg){
                this.msg = msg;
            }
    }

    protected void check(boolean expr) throws CheckFailure{
        if (!expr)
            throw new CheckFailure("Check failed");
    }
    protected void check(String msg, boolean expr)
                                throws CheckFailure{
        if (!expr)
            throw new CheckFailure(msg);
    }
}
```

The unit tester

Note that the testTheClass method is static, but creates an instance of UnitTester in the body.

We will examine each of the three auxiliary methods in turn.

```
private void setClass(String className)
    throws ClassNotFoundException,
           InstantiationException,
           IllegalAccessException {
    testClass = Class.forName(className);
    testObject = testClass.newInstance();
}
```

The method forName is as in the EncChecker.

newInstance() makes a new Object from testClass.

```
public class UnitTester{
    /**  testClass is the class that contains the
         testcases */
    private Class testClass;
    /**  testMethods is an list of the methods that
         contain the tests to be executed */
    private List testMethods;
    /** testObject is an instance of testClass. */
    private Object testObject;
    …
    public static String testTheClass(String name){
        try{
            UnitTester ut = new UnitTester();
            ut.setClass(name);
            ut.getMethods();
            return ut.performTests();
        }catch(Exception uups){
            return "Could not find class " + name;
        }
    }
}
```

Finding the methods which start with 'test'

The method is very similar to the one we used in EncCheck to find all the fields.

getMethods returns an array of all methods in the class.

Each method can be asked about its name using getName.

All methods which starts with 'test' are stored in the List testMethods.

```
private List<Method> testMethods;


private void getMethods(){
    testMethods = new ArrayList<Method>();
    try{
        Method[] allMethods = testClass.getMethods();
        for (Method m : allMethods){
            if ( m.getName().startsWith("test") ){
                testMethods.add(m);
            }
        }
    }catch(SecurityException whatWasThat){
    }
}
```

Calling the test methods

The key method here is invoke. It takes two arguments:

- the object on which to call the method (the object that will be *this* in the method call)
- an array of Objects to be given as arguments to the method.

So a method normally called as:

myObj.m(args)

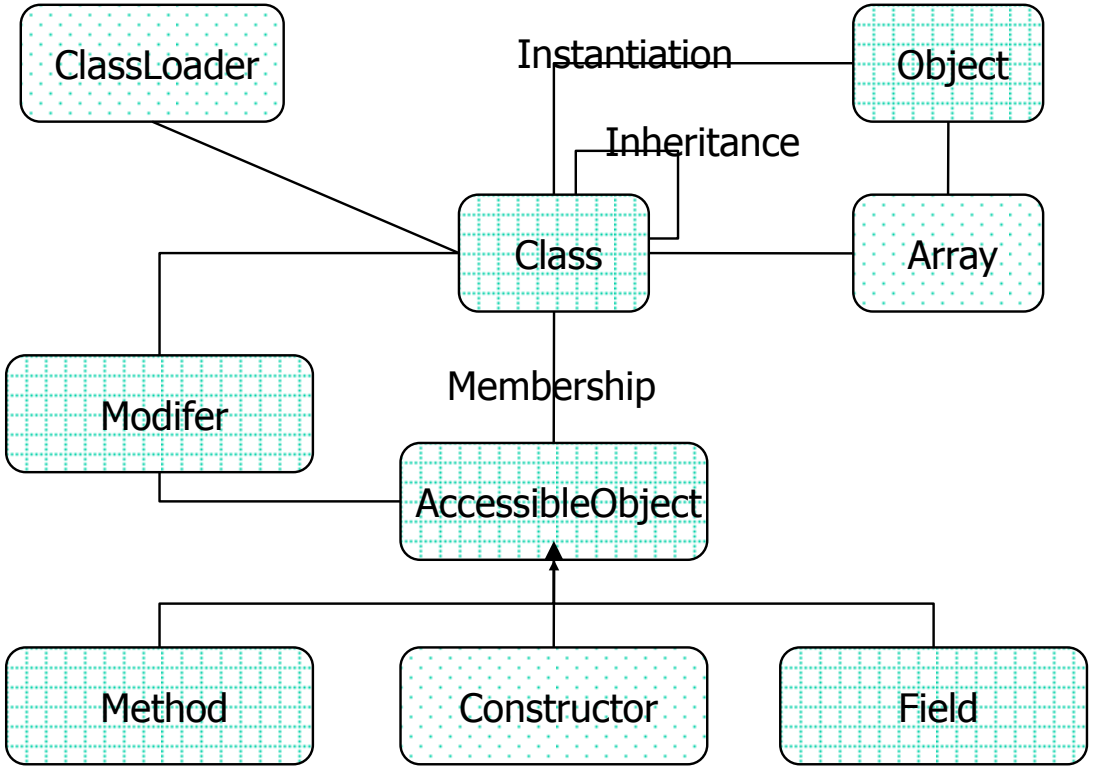
can be called using reflection as:

m.invoke(myObj, args)

instanceof is a Java operator, which returns true if the object on the left hand side is an instance of the class on the right hand side.

```
private String performTests(){
    String result = "Test of " + testClass.getName() + "\n";
    for (Method m: testMethods){
        try{
            m.invoke(testObject,new Object[0]);
            result += m.getName() + " OK\n";
        }catch(IllegalAccessException ignore){}
        catch(IllegalArgumentException ignore){}
            catch(InvocationTargetException uups){
            Throwable target = uups.getTargetException();
            if ( target instanceof TestClass.CheckFailure)
                result += m.getName() +" Check failed:
"                 + ((TestClass.CheckFailure)target).msg + "\n";
            else
                result += m.getName() + " Exception: "+
                                        target.toString() + "\n";
        }
    }
    return result;
}
```

The Java meta model



Dotted classes have not been used in these slides, squared ones have.

The class Class

The class Class represents four different types of java objects, classes, interfaces, primitive types, and array types.

One can ask a Class what kind of class it is by using the boolean methods: isArray(), isPrimitive(), isInterface() – if all return false, it is a class.

The class can tell what its super class is (getSuperClass) and which interfaces it implements (getInterfaces).

The class can tell which inner classes it has (getClasses), and an inner class can tell what class it is inner in (getDeclaringClass).

A class can tell which fields, methods, and constructors it contains.

The class Class is part of java.lang, and need not be imported to be used.

There are several ways of retrieving a Class object, i.e. the Class object for Person can be obtained using :

```
Class c1 = hans.getClass();
Class c2 = Class.forName("Person");
Class c3 = Person.class;
```

Field & AccessibleObject

The class Field represents field variables in a class, both static and non-static fields.

- Class getDeclaringClass()
- int getModifiers()
- String getName()
- Class getType()

- Object get(Object obj)
- boolean getBoolean(Object obj)
- X getX(Object obj)

- void set(Object obj, Object value)
- void setBoolean(Object obj, boolean z)
- void setX(Object obj, X d)

X can be any primitive type, but that would be too many methods to fit on a slide.

AccessibleObject is an abstract super class for Field, Method, & Constructor.

Its purpose is to allow us to mark a reflected entity to suppress access control.

- boolean isAccessible()
- void setAccessible(boolean flag)

There is a security control mechanism which can be set in the virtual machine, to prevent the manipulation of this flag.

The classes Method & Constructor

The class Method represents a method in a class.

- Class getDeclaringClass()
- Class[] getExceptionTypes()
- int getModifiers()
- String getName()
- Class[] getParameterTypes()
- Class getReturnType()
- Object invoke(Object obj, Object[] args)

The class Constructor represents a constructor in a class. It has nearly the same methods as Method:

- Class getDeclaringClass()
- Class[] getExceptionTypes()
- int getModifiers()
- String getName()
- Class[] getParameterTypes()
- Object newInstance(Object[] initargs)

The difference being that it has no return type, and that the invoke method is named newInstance.

Object Irrelevance

Object Irrelevance

```
class Car {
  ...
}
```

```
class Color {
  ...
}
```

```
class People {
  ...
}
```

Object Irrelevance (cont'd)

- Objects model the real world
- Classes form sets of those models.
- Standard Object equality comes for free.
- Structural equality must be programmed!

Problem:

- Color red1 = new Color ()
- Color green = new Color ()
- Color red2 = new Color ()

red1 and red2 are not standard equal (red1 = red2 is not true)

- User responsible for maintaining the object references.

Object Irrelevance (cont'd)

- Alternative idea: Represent objects as classes.

Interface COLOR {...}

Class Red implements COLOR;
Class Green implements COLOR;
Glass Blue implements COLOR;

- Compiler prevents you from declaring the same color twice.
- All objects of the same class are equal (thus they are irrelevant).

```
Color red1 = new Red ();
Color red2 = new Red ();
red1.eq (red2)                    results in true.
```

Annotations

Introducing annotations

- Inspecting metadata can be done using *reflection*
- Adding custom metadata can be done using *annotations*

Annotations are used by three types of tools:

- General tools
  - I.e. compilers, JavaDoc
- Specific tools
  - I.e code generators
- Introspectors
  - That access annotations at runtime using reflection

Types of annotations

There are three types of annotations:

- Marker annotations
  - Have no parameters

```
@Test
public void myTestMethod() {...}
```

- Single member annotations
  - Have a single unnamed parameter

```
@Author("John Doe")
public class MyClass {...}
```

- Normal annotations
  - Have several members

```
@Author(company="NN Inc.",
  @Name(firstname="John",

        lastname="Doe"))
interface MyInterface {...}
```

Predefined annotations

@Retention has a RetentionPolicy defining how long to retain the annotations.

Possible values are:

SOURCE
  discarded by compiler
CLASS (= default value)
  retained by compiler, not loaded by VM
RUNTIME
  retained by compiler and loaded runtime

If annotations are to be accessed using reflection the RetentionPolicy has to be set to RUNTIME.

@Target defines what types of Java elements can be annotated by this annotation.

Possible values are:

TYPE (class, interface or enum)
ANNOTATION_TYPE
CONSTRUCTOR
METHOD
PARAMETER
FIELD
LOCAL_VARIABLE
PACKAGE

Declaring annotations

A new keyword @interface is used for declaring annotations. An annotation has a name and possibly some member methods.

```
public @interface Marker { }
```

Members can be given default values using the default keyword. If no default value is given the value must be specified when the annotation is applied.

```
public @interface SingleMember {
   String value();
}
```

So all of the following are correct:

```
@MyAnno(someValue=2, otherValues={}, yetAValue="Hi" )
@MyAnno(someValue=17, otherValues={"a"})
@MyAnno(otherValues={"a","b","c"})
```

```
@Retention(value=RUNTIME)
@Target(value=METHOD)
public @interface MyAnno {
   int someValue() default 0;
   String[] otherValues();
   String yetAValue() default "[blank]";
}
```

Applying annotations

Annotations can be applied at several levels, and several different annotations can be applied at each level.

Arrays of values are entered in brackets {} with commas between the values.

```
@ClassLevelAnnotation(arg1="val1",
        arg2={"arg2.val1","arg2.val2"})
public class AnnotationExample {

    @FieldLevelAnnotation()
    public String field;

    @ConstructorLevelAnnotation()
    public AnnotationExample() {
        // code
    }

   @MethodLevelAnnotationA("val")
   @MethodLevelAnnotationB(arg1="val1“,
        arg2="val2")
    public void someMethod(String string) {
        // code
    }
}
```

Accessing annotations

Annotations can be inspected at runtime using reflection.

The classes Class, Field, Method and Constructor all have the methods:

boolean **isAnnotationPresent**(MyAnno.class) returns true if an annotation of the given annotation type exists on this element

Annotation **getAnnotation**(MyAnno.class) returns the annotation of the given annotation type if any exists on this element

Annotation[] **getDeclaredAnnotations**() Returns all annotations that are directly present on this element

Annotation[] **getAnnotations**() Returns all annotations that are present on this element, including those inherited

HelloWorldAnnotationTest

```
// The Annotation Type
@Retention(RetentionPolicy.RUNTIME)
@interface SayHi {
    public String value();
}

// The Annotated Class
@ SayHi("Hello, class!")
class HelloWorld {

    @ SayHi("Hello, field!")
    public String greetingState;

    @ SayHi("Hello, constructor!")
    public HelloWorld() {
    }

    @ SayHi("Hello, method!")
    public void greetings() {
    }
}
```

```
//The Annotation consumer
public class HelloWorldAnnotationTest
{
  public static void main (String[] args) throws Exception {
    Class<HelloWorld> clazz = HelloWorld.class;
    System.out.println( clazz.getAnnotation(SayHi.class ) );

    Constructor<HelloWorld> constructor =
          clazz.getConstructor((Class[]) null);
    System.out.println(
          constructor.getAnnotation(SayHi.class));

    Method method = clazz.getMethod("greetings");
    System.out.println(method.getAnnotation(SayHi.class));

    Field field = clazz.getField("greetingState");
    System.out.println(field.getAnnotation(SayHi.class));
  }
}
```

Other reflection capabilities not discussed

In Java

- The class loader
- Java debugging interface
- Call stack inspection
- References
- Proxy

In other languages than Java

- Changing how method calls work
- Redefining the notion of a subclass
- Redefining how object initialization is done
- Redefining how fields are accessed
- Redefining how objects are addressed

Summary

From a programming point of view, reflection does not deal with things in the application domain, but with the internals of the programs themselves.

The reflection library is for very general abstractions. Not abstractions from the application domain, but abstractions of objects.

Annotations can add custom abstractions for specific tools to use.

Use with care. It is a powerful tool. It is meant for framework development, not for application development.